Web Service Interfaces*

Dirk Beyer
EPFL, Lausanne, Switzerland
dirk.beyer@epfl.ch

Arindam Chakrabarti
University of California, Berkeley, U.S.A.
arindam@cs.berkeley.edu

Thomas A. Henzinger
EPFL, Lausanne, Switzerland & University of California, Berkeley, U.S.A.
tah@epfl.ch

ABSTRACT

We present a language for specifying web service interfaces. A web service interface puts three kinds of constraints on the users of the service. First, the interface specifies the methods that can be called by a client, together with types of input and output parameters; these are called *signature constraints*. Second, the interface may specify propositional constraints on method calls and output values that may occur in a web service conversation; these are called *consistency constraints*. Third, the interface may specify temporal constraints on the ordering of method calls; these are called *protocol constraints*. The interfaces can be used to check, first, if two or more web services are compatible, and second, if a web service A can be safely substituted for a web service B. The algorithm for *compatibility checking* verifies that two or more interfaces fulfill each others' constraints. The algorithm for *substitutivity checking* verifies that service A demands fewer and fulfills more constraints than service B.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*; D.2.12 [**Software Engineering**]: Interoperability—*Interface definition languages*

General Terms

Design, Reliability, Verification

Keywords

Web services, Web service interfaces, Web service compatibility, Web service substitutivity, Formal specification, Formal verification

*This research was supported in part by the ONR grant N00014-02-1-0671 and by the NSF grants CCR-0234690 and CCR-0225610.

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

1. INTRODUCTION

Interface formalisms are used to avoid errors in component-based system design. In particular, by checking at design-time if two or more interfaces are compatible, we can ensure that the corresponding components work together properly at run-time. For example, programming-language types offer a simple interface formalism: it can be checked at compile-time if the number and types of the parameters of a function call and the function definition match. Richer interface formalisms have been devised for software components whose correct interaction depends on communication protocols [7], timing [9], or resource use [5].

Web services offer a particularly natural and important application domain for interface formalisms. A web service often depends on other web services, which have been implemented by different vendors, and their correct usage is governed by rules. Such rules may constrain data types, but they may also express temporal constraints (e.g., "a shipping order should be placed only if a credit card check has succeeded"). We present a language for equipping web services with interfaces that formalize such rules. The benefits of our language are two-fold. First, it can be checked algorithmically if two or more interfaces fulfill each other's rules; in this case the interfaces are called *compatible*. Second, it can be checked algorithmically if one interface can be safely replaced by another one; in this case the latter interface is said to *refine* the former.

Compatibility and refinement checks, like model checking, reduce the burden on testing for system integration and validation. However, we explicitly propose to use formal methods to specify and verify the *interfaces* of web services, not their implementations (for implementation checking see, e.g., [13, 15, 11, 20]). Interface checking stands a much better chance of succeeding in practice than implementation checking, as interfaces are usually less complex than the corresponding implementations. Indeed, good interface design suggests that an interface exposes all information about a web service that is needed to use the service properly, and that the interface does not expose more than that. This is why we offer not one, but three interface languages, which are increasingly more expressive.

The first language, called *web service signatures*, exposes only the names and types of externally callable methods and their return values. For example, a web service signature may offer the method `credit_card_check` with the two return values `success` and `failure`. The second language, called *consistency interfaces*, equips web service signatures with propositional constraints on the

consistency between various method calls and return values. For example, it may be inconsistent to have both $\langle$`credit_card_check`, `failure`$\rangle$ and $\langle$`ship_order`, `success`$\rangle$ in the same web service conversation. The third and richest language, called *protocol interfaces*, equips web service signatures with temporal constraints between method calls. For example, a protocol interface may prohibit conversations where $\langle$`ship_order`, `success`$\rangle$ occurs before $\langle$`credit_card_check`, `success`$\rangle$.

The compatibility and refinement of web service signatures can be type checked. Consistency interfaces are stateless; their compatibility and refinement can be checked by solving propositional (boolean) constraints. Protocol interfaces are stateful; they can be naturally expressed by automata (state machines) with nondeterminism, recursion, and thread creation. Their compatibility can be checked by model checking temporal safety constraints.

2. WEB SERVICE SIGNATURES

Let $\mathcal{M}$ be a finite set of *web methods* and $\mathcal{O}$ be a finite set of *outcomes*. The outcome is used to encode a return value from the web method, or other behavioral differences between various calls to the web method; for instance, if the invocation will lead to a call to a callback method or not. An *action* a on $\mathcal{M}$ and $\mathcal{O}$ is a pair $\langle m, o\rangle \in \mathcal{M} \times \mathcal{O}$. An action $\langle m, o\rangle$ constitutes an *"assumption"* at the invocation point that the call to web method m will *eventually* lead to the outcome o for this call. This coupling of the call together with its eventual outcome is found to be very useful in reasoning about the behavior of web services. The set of actions on $\mathcal{M}$ and $\mathcal{O}$ is denoted by $\mathcal{A} \subseteq \mathcal{M} \times \mathcal{O}$.

A *web service signature* $\mathcal{S}$ is a partial function $\mathcal{S} : \mathcal{A} \to 2^{\mathcal{A}}$, which assigns to an action a a set of actions which can be invoked by a. A web service signature $\mathcal{S}$ *supports* an action $a \in \mathcal{A}$ if $\mathcal{S}(a)$ is defined; $\mathcal{S}$ *supports* a web method $m \in \mathcal{M}$ if some action $\langle m, o\rangle$ is supported by $\mathcal{S}$. An action a *requires* an action a' in $\mathcal{S}$ if $a' \in \mathcal{S}(a)$. A web service signature $\mathcal{S}$ *requires* an action a' if some action a requires action a' in $\mathcal{S}$. A web service signature $\mathcal{S}$ is *well-formed* if for every web method m supported by $\mathcal{S}$, every action $\langle m, o\rangle \in \mathcal{A}$ required by $\mathcal{S}$ is supported by $\mathcal{S}$. In the rest of the paper, the word "signature" stands for "well-formed signature".

Intuitively, a signature element $(\langle m, o\rangle, S)$ says that when the web method m is called, and the caller assumes the outcome o, this signature pledges to support this action, and itself relies on that the assumptions carried by the actions $a' \in S$ are fulfilled (by either this signature, or by the environment). Thus, a web service signature relates the *"guarantees"* made (actions supported) by the interface to the *"assumptions"* (actions assumed to be supported, either by the interface itself or by the environment) under which they are made. The signature is well-formed if the assumptions and guarantees are mutually consistent.

EXAMPLE 1. **(Supply chain management application)** We briefly describe a supply chain management application [2] that we will use as a running example in the rest of the paper to illustrate concepts as we introduce them. The application consists of five web services: `Shop`, `Store`, `Bank`, `Transport`, and `Supplier`. We will present interface models for `Shop` and `Store` in the following sections. We will focus on particular aspects of the services and not provide a full-fledged model.

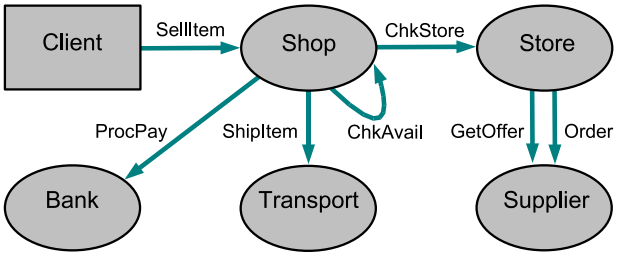

Figure 1: The supply chain management application

Figure 1 gives an overview of the interactions between the web services. The `Shop` interface supports the web method `SellItem` which is called by the Client application to start the selling process, and it supports `ChkAvail` which is a subroutine (called by `Shop` itself) that checks availability of items for sale. `Shop` requires the web method `ChkStore` to be implemented by `Store` to check whether the desired items are available in stock. It also requires `ShipItem` to be implemented by `Transport` to ship the sold items to the customer, and it requires `ProcPay` to be implemented by `Bank` to process credit card payments. `Store` requires two web methods to be implemented by `Supplier`: `GetOffer` and `Order`, to get an offer for an item, and to order new items if the stock falls below a certain threshold. ∎

EXAMPLE 2. **(Well-formed signature)** To model the `Shop` we need the following sets of methods, outcomes and actions, before we can define a web service signature for it:

$\mathcal{M} = \{$`SellItem`, `ChkAvail`, `ChkStore`, `ProcPay`, `ShipItem`, `GetOffer`, `Order`$\}$
$\mathcal{O} = \{$`SOLD`, `NOTFOUND`, `OK`, `FAIL`, `REC`$\}$
$\mathcal{A} = \{\langle$`SellItem`, `SOLD`$\rangle$, $\langle$`SellItem`, `FAIL`$\rangle$, $\langle$`SellItem`, `NOTFOUND`$\rangle$, $\langle$`ChkAvail`, `OK`$\rangle$, $\langle$`ChkAvail`, `FAIL`$\rangle$, $\langle$`ChkStore`, `OK`$\rangle$, $\langle$`ChkStore`, `FAIL`$\rangle$, $\langle$`ProcPay`, `OK`$\rangle$, $\langle$`ProcPay`, `FAIL`$\rangle$, $\langle$`ShipItem`, `OK`$\rangle$, $\langle$`ShipItem`, `FAIL`$\rangle$, $\langle$`GetOffer`, `REC`$\rangle$, $\langle$`Order`, `OK`$\rangle\}$

The `Shop` interface can be modeled as the following web service signature:

$\mathcal{S}_{\text{Shop}} = \{$
$\langle$`SellItem`, `SOLD`$\rangle \mapsto$ $\{\langle$`ChkAvail`, `OK`$\rangle$, $\langle$`ProcPay`, `OK`$\rangle$, $\langle$`ShipItem`, `OK`$\rangle\}$
$\langle$`SellItem`, `FAIL`$\rangle \mapsto$ $\{\langle$`ChkAvail`, `OK`$\rangle$, $\langle$`ChkAvail`, `FAIL`$\rangle$, $\langle$`ProcPay`, `OK`$\rangle$, $\langle$`ProcPay`, `FAIL`$\rangle$, $\langle$`ShipItem`, `FAIL`$\rangle\}$
$\langle$`ChkAvail`, `OK`$\rangle \mapsto$ $\{\langle$`ChkStore`, `OK`$\rangle\}$
$\langle$`ChkAvail`, `FAIL`$\rangle \mapsto$ $\{\langle$`ChkStore`, `FAIL`$\rangle\}$
$\}$

For instance, action $\langle$`SellItem`, `SOLD`$\rangle$ is supported by $\mathcal{S}_{\text{Shop}}$, and actions $\langle$`ChkAvail`, `OK`$\rangle$, $\langle$`ProcPay`, `OK`$\rangle$ and $\langle$`ShipItem`, `OK`$\rangle$ are assumed to be supported by the environment. Signature $\mathcal{S}_{\text{Shop}}$ is well-formed, because it supports all possible actions for its supported methods. ∎

Compatibility and composition. For two web service interfaces, we want to check whether they can cooperate properly, i.e., whether their signatures mutually fulfill their guarantees and assumptions.

Given two web service signatures $\mathcal{S}_1$ and $\mathcal{S}_2$ with $\text{dom}(\mathcal{S}_1) \cap \text{dom}(\mathcal{S}_2) = \emptyset$, if $\mathcal{S}_c = \mathcal{S}_1 \cup \mathcal{S}_2$ is well-formed, then $\mathcal{S}_1$ and $\mathcal{S}_2$ are *compatible* (denoted by $comp(\mathcal{S}_1, \mathcal{S}_2)$), and their *composition* (denoted by $\mathcal{S}_1 \parallel \mathcal{S}_2$) is $\mathcal{S}_c$. The composition operation is commutative and associative. Compatibility and composition of web service signatures can be computed in linear time.

Composing web services allows us to reason about the behavior exhibited when, for instance, a particular airline reservation service $\mathcal{S}_a$ and a particular hotel reservation service $\mathcal{S}_h$ are used as part of a larger system. Note that the composition is still an open system. For instance, $\mathcal{S}_a$ and $\mathcal{S}_h$ may both use a credit card processing service $\mathcal{S}_c$. Assumptions made about the environment are allowed; the goal is to check whether an environment fulfilling such assumptions can exist. The airline ticket reservation service and the hotel room reservation service may make assumptions about the behavior they expect of each other, and additionally about the credit card processing service in their environment, which they both use. Our formalism considers the airline and hotel reservation web services compatible as long as their mutual behavioral assumptions and guarantees are consistent, assuming that a credit card processing service fulfilling all expected requirements can exist. Intuitively, this is why our formalism is an *interface theory* [8].

EXAMPLE 3. **(Compatibility of signatures)** Let us build the composition of the signature $\mathcal{S}_{\texttt{Shop}}$ with the following signature for `Store`:

$$
\begin{aligned}
&\mathcal{S}_{\texttt{Store}} = \{\\
&\ \langle \texttt{ChkStore}, \texttt{OK}\rangle \mapsto \quad \{\langle \texttt{GetOffer}, \texttt{OK}\rangle, \langle \texttt{Order}, \texttt{OK}\rangle\}\\
&\ \langle \texttt{ChkStore}, \texttt{FAIL}\rangle \mapsto \quad \{\langle \texttt{GetOffer}, \texttt{OK}\rangle, \langle \texttt{Order}, \texttt{OK}\rangle\}\\
&\}
\end{aligned}
$$

Both actions require two other actions supported by `Supplier` to order new items if the stock is below a certain threshold. Signature $\mathcal{S}_{\texttt{Store}}$ is well-formed. The union $\mathcal{S}_{\texttt{Shop}} \cup \mathcal{S}_{\texttt{Store}}$ is also well-formed because it supports all actions for its supported methods and no action is supported in both signatures. Therefore the signatures are compatible and their composition is $\mathcal{S}_{\texttt{Shop}} \cup \mathcal{S}_{\texttt{Store}}$. ∎

Substitutivity. To enable top-down design, we want to be able to replace a component embedded as part of a larger system (its environment) with a new component, while ensuring that the entire system still cooperates properly as before. Intuitively, we need to ensure that the replacement component behaves similarly to the replaced one as far as the environment assumptions and guarantees are concerned.

Given two web service signatures $\mathcal{S}$ and $\mathcal{S}'$, we say $\mathcal{S}'$ *refines* $\mathcal{S}$ (written $\mathcal{S}' \preccurlyeq \mathcal{S}$) if

i) for every $a \in \mathcal{A}$, if $\mathcal{S}$ supports a, then $\mathcal{S}'$ supports a,

ii) for every $a, a' \in \mathcal{A}$, if $\mathcal{S}$ supports a, and a requires a' in $\mathcal{S}'$, then a requires a' in $\mathcal{S}$, and

iii) for all web methods m not supported by $\mathcal{S}'$, if $\mathcal{S}'$ requires an action $\langle m, o\rangle$, then $\mathcal{S}$ requires $\langle m, o\rangle$.

The first condition ensures that the refined web service signature *"guarantees"* to support every action supported by the abstract one. The other two conditions ensure that the refined web service signature does not *"assume"* additional actions to be supported by the environment under situations where it is used exactly as the abstract one would have been used. Given two web service signatures $\mathcal{S}$ and $\mathcal{S}'$, the question if $\mathcal{S}' \preccurlyeq \mathcal{S}$ can be decided in linear time.

EXAMPLE 4. **(Substitutivity of signatures)** Let $\mathcal{S}'_{\texttt{Shop}}$ be a second web service signature defined as

$$
\begin{aligned}
&\mathcal{S}'_{\texttt{Shop}} = \{\\
&\ \langle \texttt{SellItem}, \texttt{SOLD}\rangle \mapsto \quad \{\langle \texttt{ChkAvail}, \texttt{OK}\rangle, \langle \texttt{ProcPay}, \texttt{OK}\rangle\}\\
&\ \langle \texttt{SellItem}, \texttt{FAIL}\rangle \mapsto \quad \{\langle \texttt{ChkAvail}, \texttt{OK}\rangle, \langle \texttt{ProcPay}, \texttt{OK}\rangle,\\
&\qquad\qquad\qquad\ \langle \texttt{ProcPay}, \texttt{FAIL}\rangle\}\\
&\ \langle \texttt{SellItem}, \texttt{NOTFOUND}\rangle \mapsto \quad \{\langle \texttt{ChkAvail}, \texttt{FAIL}\rangle\}\\
&\ \langle \texttt{ChkAvail}, \texttt{OK}\rangle \mapsto \quad \{\langle \texttt{ChkStore}, \texttt{OK}\rangle\}\\
&\ \langle \texttt{ChkAvail}, \texttt{FAIL}\rangle \mapsto \quad \{\langle \texttt{ChkStore}, \texttt{FAIL}\rangle\}\\
&\}
\end{aligned}
$$

Compared with $\mathcal{S}_{\texttt{Shop}}$, the signature $\mathcal{S}'_{\texttt{Shop}}$ supports an additional action $\langle \texttt{SellItem}, \texttt{NOTFOUND}\rangle$ and requires no action for shipping (e.g., because it implements the shipping functionality itself). It is a refinement of $\mathcal{S}_{\texttt{Shop}}$ because it i) supports all actions which $\mathcal{S}_{\texttt{Shop}}$ supports, ii) these "refined" actions require only a subset of the actions which are required by the actions in $\mathcal{S}_{\texttt{Shop}}$, iii) the new action requires only actions which are already required by some action in $\mathcal{S}_{\texttt{Shop}}$. ∎

3. CONSISTENCY INTERFACES

Let $\mathcal{B}(P)$ be the set of expressions over the set of propositions P, using the binary operators $\sqcap$ and $\sqcup$, and the propositional constant $\top$. Our language of expressions has some important properties in common with the language of boolean expressions, but there are some essential differences which we will point out as appropriate. We use the symbols $\sqcup$ and $\sqcap$ instead of $\vee$ and $\wedge$ to reflect the similarity, and the difference.

Given a set of actions $\mathcal{A}$, a *consistency interface* $\mathcal{C}$ is a partial function $\mathcal{C} : \mathcal{A} \to \mathcal{B}(\mathcal{A})$. The element $(\langle m, o\rangle, \top) \in \mathcal{C}$ means that interface $\mathcal{C}$ supports a method m, and when m is called, and the caller expects the outcome to be o, the interface $\mathcal{C}$ guarantees to fulfill the expectation, since $\mathcal{C}(\langle m, o\rangle)$ is defined. Further, $\mathcal{C}$ provides this guarantee making the assumption $\top$, which is always fulfilled, i.e., action $\langle m, o\rangle$ is supported under no assumptions. Intuitively, this means that the code executed when web method m is called can lead to the outcome o without calling any other web method. The element $(a, a') \in \mathcal{C}$, where $a' = \langle m', o'\rangle$, says that the action a is supported by $\mathcal{C}$, and when a is invoked, $\mathcal{C}$ calls m' in turn, and expects the outcome o'. The element $(a, \varphi_1 \sqcap \varphi_2) \in \mathcal{C}$ says that action a is supported by $\mathcal{C}$, and leads to the combination of the two sets of conversations, one represented by φ_1 and one represented by φ_2. The element $(a, \varphi_1 \sqcup \varphi_2) \in \mathcal{C}$ says that action a is supported by $\mathcal{C}$, but can lead to the set of conversations represented by φ_1, or to the set of conversations represented by φ_2, nondeterministically chosen by the interface. Allowing nondeterminism in interfaces allows us to make them abstract; i.e., the interface can forget about the implementation details of the web service code, and only capture its external behavior in terms of patterns of the web method calls it makes.

Given a consistency interface $\mathcal{C}$, the *underlying web service signature* of $\mathcal{C}$ (denoted by $sig(\mathcal{C})$) is defined as follows: for all $a \in \mathcal{A}$, if $\mathcal{C}(a)$ is defined, then $sig(\mathcal{C})(a) = \{a' \mid a' \text{ occurs in } \mathcal{C}(a)\}$, and $sig(\mathcal{C})(a)$ is undefined otherwise. A consistency interface $\mathcal{C}$ is *well-formed* if $sig(\mathcal{C})$ is well-formed.

In the rest of the paper, "consistency interface" stands for "well-formed consistency interface".

EXAMPLE 5. **(Well-formed consistency interface)** Let us model the `Shop` web service as consistency interface:

$$
\begin{array}{ll}
\mathcal{C}_{\texttt{Shop}} = \{ & \\
\ \langle \texttt{SellItem}, \texttt{SOLD} \rangle \mapsto & \langle \texttt{ChkAvail}, \texttt{OK} \rangle \sqcap \langle \texttt{ProcPay}, \texttt{OK} \rangle \\
& \sqcap \langle \texttt{ShipItem}, \texttt{OK} \rangle \\
\ \langle \texttt{SellItem}, \texttt{FAIL} \rangle \mapsto & \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \sqcup (\langle \texttt{ChkAvail}, \texttt{OK} \rangle \\
& \sqcap (\langle \texttt{ProcPay}, \texttt{FAIL} \rangle \sqcup \\
& (\langle \texttt{ProcPay}, \texttt{OK} \rangle \sqcap \langle \texttt{ShipItem}, \texttt{FAIL} \rangle) \\
&)) \\
\ \langle \texttt{ChkAvail}, \texttt{OK} \rangle \mapsto & \langle \texttt{ChkStore}, \texttt{OK} \rangle \\
\ \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \mapsto & \langle \texttt{ChkStore}, \texttt{FAIL} \rangle \\
\} &
\end{array}
$$

For action $\langle \texttt{SellItem}, \texttt{SOLD} \rangle$, all three actions occur together. For action $\langle \texttt{SellItem}, \texttt{FAIL} \rangle$, action $\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle$ occurs alone, or action $\langle \texttt{ChkAvail}, \texttt{OK} \rangle$ occurs together with either action $\langle \texttt{ProcPay}, \texttt{FAIL} \rangle$ or both, actions $\langle \texttt{ProcPay}, \texttt{OK} \rangle$ and $\langle \texttt{ShipItem}, \texttt{FAIL} \rangle$. Note that nothing is said about the order of their occurrence. The actions for method `ChkAvail` result in calls of the method `ChkStore` in `Store` (delegation of service).

The underlying signature $sig(\mathcal{C}_{\texttt{Shop}})$ of $\mathcal{C}_{\texttt{Shop}}$ is $\mathcal{S}_{\texttt{Shop}}$ from Example 2, which is a well-formed signature, and hence $\mathcal{C}_{\texttt{Shop}}$ is a well-formed consistency interface. ∎

3.1 Compatibility and Composition

Intuitively, given two consistency interfaces, if their mutual behavioral assumptions and guarantees allow them to co-operate, we say they are *compatible*, and we can *compose* them to a single consistency interface, in which some of their internal details are forgotten and the combined behavioral assumptions and guarantees of interest for the environment are retained.

Given two consistency interfaces $\mathcal{C}_1$ and $\mathcal{C}_2$, if the underlying signatures $sig(\mathcal{C}_1)$ and $sig(\mathcal{C}_2)$ are compatible, then $\mathcal{C}_1$ and $\mathcal{C}_2$ are *compatible* (denoted by $comp(\mathcal{C}_1, \mathcal{C}_2)$), and their *composition* (denoted $\mathcal{C}_1 \parallel \mathcal{C}_2$) is $\mathcal{C}_1 \cup \mathcal{C}_2$. The composition operation is commutative and associative. Compatibility and composition of consistency interfaces can be computed in linear time.

EXAMPLE 6. **(Compatibility of consistency interfaces)** Consider the following consistency interface for `Store`:

$$
\begin{array}{ll}
\mathcal{C}_{\texttt{StoreF1}} = \{ & \\
\ \langle \texttt{ChkStore}, \texttt{OK} \rangle \mapsto & \{\mathrm{T} \sqcup (\langle \texttt{GetOffer}, \texttt{REC} \rangle \sqcap \langle \texttt{Order}, \texttt{OK} \rangle)\} \\
\} &
\end{array}
$$

When action $\langle \texttt{ChkStore}, \texttt{OK} \rangle$ is invoked, `Store` does nothing, or gets offers and orders more items (if the stock is below threshold). Note that nondeterminism allows the stock to be abstracted away.

Let us try to compose the `Shop` and `Store` interfaces. Are they compatible? No, because their union $\mathcal{C}_c$ is not a well-formed interface; because $\mathcal{C}_c$ supports an action $\langle \texttt{ChkStore}, \texttt{OK} \rangle$ and thus supports method `ChkStore`, but requires an action $\langle \texttt{ChkStore}, \texttt{FAIL} \rangle$ that is not supported. Intuitively, this means that `Shop` and `Store` do not agree on their mutual guarantees and assumptions, and therefore cannot work with each other.

The following consistency interface fixes the bug of $\mathcal{C}_{\texttt{StoreF1}}$ by supporting action $\langle \texttt{ChkStore}, \texttt{FAIL} \rangle$:

$$
\begin{array}{ll}
\mathcal{C}_{\texttt{Store}} = \{ & \\
\ \langle \texttt{ChkStore}, \texttt{OK} \rangle \mapsto & \{\mathrm{T} \sqcup (\langle \texttt{GetOffer}, \texttt{OK} \rangle \sqcap \langle \texttt{Order}, \texttt{OK} \rangle)\} \\
\ \langle \texttt{ChkStore}, \texttt{FAIL} \rangle \mapsto & \{\langle \texttt{GetOffer}, \texttt{OK} \rangle \sqcap \langle \texttt{Order}, \texttt{OK} \rangle\} \\
\} &
\end{array}
$$

The action $\langle \texttt{ChkStore}, \texttt{OK} \rangle$ is the same as in $\mathcal{C}_{\texttt{StoreF}}$, and for action $\langle \texttt{ChkStore}, \texttt{FAIL} \rangle$ it orders new items. The consistency interfaces $\mathcal{C}_{\texttt{Shop}}$ and $\mathcal{C}_{\texttt{Store}}$ are compatible, and can be composed to $\mathcal{C}_{\texttt{Shop}} \parallel \mathcal{C}_{\texttt{Store}}$. ∎

3.2 Specifications

The notion of well-formedness removes a general class of errors in consistency interfaces. However, depending on the particular application in which a service will be used, additional properties would be required of it. We identify a class of safety properties that seem the most useful in practice and propose a property specification language and a verification scheme as follows.

In the context of consistency interfaces, a *conversation* is a set of actions that are exhibited together. Sets of conversations are concisely represented using expressions $\varphi \in \mathcal{B}(\mathcal{A})$. Intuitively, the expression T represents the empty conversation. The expression a (where a is a supported action) represents the set of possible conversations exhibited when action a is invoked. If a is not supported, then the expression a represents the set consisting of only the conversation $\{a\}$. Thus, we postpone judgement about the behavior exhibited by an action a until enough information is available about the actions a requires. Intuitively, this is why our formalism allows systems to be developed and analyzed incrementally; as long as some environment exists in which the (open) system under development can operate correctly, it will be considered to be correct. The expression $\varphi_1 \sqcup \varphi_2$ represents a nondeterministic choice between φ_1 and φ_2, and hence represents the set consisting of all conversations represented by φ_1 or φ_2. The expression $\varphi_1 \sqcap \varphi_2$ represents the combination of the two sets of conversations, i.e., the actions of each pair of conversations, one each from φ_1 and φ_2, can be exhibited together in a conversation of $\varphi_1 \sqcap \varphi_2$.

The set of conversations represented by an expression from $\mathcal{B}(\mathcal{A})$ is defined by the function $[\![.]\!] : \mathcal{B}(\mathcal{A}) \to 2^{2^{\mathcal{A}}}$, which is inductively defined as the least solution of the following system of equations:

$$
\begin{array}{lcll}
[\![\mathrm{T}]\!] & = & \{\{\}\} & \\
[\![a]\!] & = & \{\{a\} \cup y \mid y \in [\![\mathcal{C}(a)]\!]\} & \text{if } a \in \mathrm{dom}(\mathcal{C}) \\
[\![a]\!] & = & \{\{a\}\} & \text{if } \neg a \in \mathrm{dom}(\mathcal{C}) \\
[\![\varphi_1 \sqcup \varphi_2]\!] & = & [\![\varphi_1]\!] \cup [\![\varphi_2]\!] & \\
[\![\varphi_1 \sqcap \varphi_2]\!] & = & \{x \cup y \mid x \in [\![\varphi_1]\!], y \in [\![\varphi_2]\!]\} &
\end{array}
$$

where $a \in \mathcal{A}$ and $\varphi_1, \varphi_2 \in \mathcal{B}(\mathcal{A})$.

Note that according to the above definition, the operators $\sqcup$ and $\sqcap$ are commutative and associative, and $\sqcap$ distributes over $\sqcup$, thus acting like the boolean $\vee$ and $\wedge$ operators respectively. However, while $\sqcup$ is idempotent (like $\vee$), $\sqcap$ is not (unlike $\wedge$); and consequently $\sqcup$ does not distribute over $\sqcap$. The following example illustrates the idea concretely.

EXAMPLE 7. **(Idempotence)** The expression $\varphi \sqcup \varphi$ represents choice between two completely equivalent alternatives, so $\sqcup$ must be idempotent. Now consider the expression $\varphi = (a \sqcup b) \sqcap (a \sqcup b)$. It represents two duplicate

invocations of $(a \sqcup b)$, which make independent choices and hence φ exhibits a richer set of conversations than a single invocation of $(a \sqcup b)$. Formally, $[\![(a \sqcup b) \sqcap (a \sqcup b)]\!] = \{\{a\},\{a,b\},\{b\}\} \neq \{\{a\},\{b\}\} = [\![a \sqcup b]\!]$. ∎

A *specification* ψ for a consistency interface is a formula $a \not\rightsquigarrow S$ where $a \in \mathcal{A}$ and $S \subseteq \mathcal{A}$. Intuitively, a specification represents the property that the invocation of action a must not lead to a conversation in which the actions in set S are all exhibited together. Formally, a specification $\psi = a \not\rightsquigarrow S$ *is satisfied by* a consistency interface $\mathcal{C}$ (denoted $\mathcal{C} \models \psi$) if $S \not\subseteq y$ for all $y \in [\![\mathcal{C}(a)]\!]$. The specification satisfaction problem for consistency interfaces is in co-NP.

Given two compatible consistency interfaces $\mathcal{C}_1$ and $\mathcal{C}_2$, and a specification ψ, then the following holds: $(\mathcal{C}_1 \parallel \mathcal{C}_2) \models \psi \;\Rightarrow\; (\mathcal{C}_1 \models \psi \wedge \mathcal{C}_2 \models \psi)$. The converse is not true.

EXAMPLE 8. **(Specification for consistency interfaces)** Consider the interface $\mathcal{C}_c = \mathcal{C}_{\texttt{Shop}} \parallel \mathcal{C}_{\texttt{Store}}$ and the following specification ψ:

$\langle \texttt{SellItem}, \texttt{FAIL} \rangle \not\rightsquigarrow \{\langle \texttt{ChkStore}, \texttt{FAIL} \rangle, \langle \texttt{ProcPay}, \texttt{OK} \rangle\}$

The specification requires that the client must not be charged for an item which is not available at the store. To check whether $\mathcal{C}_c$ satisfies ψ, we have to compute $[\![\mathcal{C}_c(\langle \texttt{SellItem}, \texttt{FAIL} \rangle)]\!]$, i.e.,

$[\![\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \sqcup (\langle \texttt{ChkAvail}, \texttt{OK} \rangle \sqcap (\langle \texttt{ProcPay}, \texttt{FAIL} \rangle \sqcup (\langle \texttt{ProcPay}, \texttt{OK} \rangle \sqcap \langle \texttt{ShipItem}, \texttt{FAIL} \rangle)))]\!]$,

which is the following set (of sets):

$$\{\{\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle, \langle \texttt{ChkStore}, \texttt{FAIL} \rangle, \langle \texttt{GetOffer}, \texttt{OK} \rangle, \langle \texttt{Order}, \texttt{OK} \rangle\},$$
$$\{\langle \texttt{ChkAvail}, \texttt{OK} \rangle, \langle \texttt{ChkStore}, \texttt{OK} \rangle, \langle \texttt{ProcPay}, \texttt{FAIL} \rangle\},$$
$$\{\langle \texttt{ChkAvail}, \texttt{OK} \rangle, \langle \texttt{ChkStore}, \texttt{OK} \rangle, \langle \texttt{GetOffer}, \texttt{OK} \rangle, \langle \texttt{Order}, \texttt{OK} \rangle, \langle \texttt{ProcPay}, \texttt{FAIL} \rangle\},$$
$$\{\langle \texttt{ChkAvail}, \texttt{OK} \rangle, \langle \texttt{ChkStore}, \texttt{OK} \rangle, \langle \texttt{ProcPay}, \texttt{OK} \rangle, \langle \texttt{ShipItem}, \texttt{FAIL} \rangle\}$$
$$\{\langle \texttt{ChkAvail}, \texttt{OK} \rangle, \langle \texttt{ChkStore}, \texttt{OK} \rangle, \langle \texttt{GetOffer}, \texttt{OK} \rangle, \langle \texttt{Order}, \texttt{OK} \rangle, \langle \texttt{ProcPay}, \texttt{OK} \rangle, \langle \texttt{ShipItem}, \texttt{FAIL} \rangle\}$$
$$\}$$

We observe that there is no y in $[\![\mathcal{C}_c(\langle \texttt{SellItem}, \texttt{FAIL} \rangle)]\!]$ such that $\{\langle \texttt{ChkStore}, \texttt{FAIL} \rangle, \langle \texttt{ProcPay}, \texttt{OK} \rangle\} \subseteq y$, and therefore $(\mathcal{C}_{\texttt{Shop}} \parallel \mathcal{C}_{\texttt{Store}}) \models \psi$. ∎

EXAMPLE 9. **(Non-recursive conversation)** Consider now the following interface for the `Shop` web service:

$$\begin{aligned}
&\mathcal{C}'_{\texttt{Shop}} = \{\\
&\quad \langle \texttt{SellItem}, \texttt{SOLD} \rangle \mapsto \langle \texttt{ChkAvail}, \texttt{OK} \rangle \sqcap \langle \texttt{ProcPay}, \texttt{OK} \rangle\\
&\quad \langle \texttt{SellItem}, \texttt{FAIL} \rangle \mapsto \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \sqcup (\\
&\qquad\qquad \langle \texttt{ChkAvail}, \texttt{OK} \rangle \sqcap \langle \texttt{ProcPay}, \texttt{FAIL} \rangle)\\
&\quad \langle \texttt{SellItem}, \texttt{NOTFOUND} \rangle \mapsto \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle\\
&\quad \langle \texttt{ChkAvail}, \texttt{OK} \rangle \mapsto \langle \texttt{ChkStore}, \texttt{OK} \rangle\\
&\quad \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \mapsto \langle \texttt{ChkStore}, \texttt{FAIL} \rangle\\
&\}
\end{aligned}$$

Let us compose $\mathcal{C}'_{\texttt{Shop}}$ with the following version of `Store`:

$$\begin{aligned}
&\mathcal{C}_{\texttt{StoreF2}} = \{\\
&\quad \langle \texttt{ChkStore}, \texttt{OK} \rangle \mapsto \top \sqcup (\langle \texttt{GetOffer}, \texttt{OK} \rangle \sqcap \langle \texttt{Order}, \texttt{OK} \rangle)\\
&\quad \langle \texttt{ChkStore}, \texttt{FAIL} \rangle \mapsto \langle \texttt{SellItem}, \texttt{NOTFOUND} \rangle \sqcap\\
&\qquad\qquad \langle \texttt{GetOffer}, \texttt{OK} \rangle \sqcap \langle \texttt{Order}, \texttt{OK} \rangle\\
&\}
\end{aligned}$$

Whenever method `ChkStore` reports that there are no more items available (by action $\langle \texttt{ChkStore}, \texttt{FAIL} \rangle$), the `Store` interface notifies the `Shop` interface that there are no more items available. We can check whether the conversations of these two web services contain a recursive invocation of $\langle \texttt{ChkStore}, \texttt{FAIL} \rangle$ by checking the following specification on the composed interface:

$\psi_R = \langle \texttt{ChkStore}, \texttt{FAIL} \rangle \not\rightsquigarrow \{\langle \texttt{ChkStore}, \texttt{FAIL} \rangle\}$

The property does not hold because $\langle \texttt{ChkStore}, \texttt{FAIL} \rangle$ invokes $\langle \texttt{SellItem}, \texttt{NOTFOUND} \rangle$ which invokes $\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle$, and this action invokes $\langle \texttt{ChkStore}, \texttt{FAIL} \rangle$ again. ∎

3.3 Substitutivity

While implementing a web service application, a developer often wants to use an off-the-shelf service $\mathcal{P}'$ to implement some desired functionality for which she has an abstract place-holder $\mathcal{P}$ in her overall design. In such situations, it is required to decide if $\mathcal{P}'$ fulfills the behavioral requirements specified in $\mathcal{P}$, to avoid errors in the overall design. Intuitively, *refinement* captures the notion of one service being substitutable in place of another.

Given two consistency interfaces $\mathcal{C}$ and $\mathcal{C}'$, we say $\mathcal{C}'$ *refines* $\mathcal{C}$ (written $\mathcal{C}' \preccurlyeq \mathcal{C}$) if

i) $sig(\mathcal{C}') \preccurlyeq sig(\mathcal{C})$, and

ii) for every $a \in \mathcal{A}$, if $\mathcal{C}$ supports a, then for all conversations $y \in [\![\mathcal{C}'(a)]\!]$, there exists a conversation $x \in [\![\mathcal{C}(a)]\!]$ such that $y \subseteq x$.

The definition above allows the refinement $\mathcal{C}'$ to drop conversations, or actions from a conversation, for actions supported by $\mathcal{C}$; $\mathcal{C}'$ is also allowed to support additional actions that $\mathcal{C}$ does not, but it is not allowed to require additional actions; it can implement actions a not supported by $\mathcal{C}$ only by requiring actions b already required by $\mathcal{C}$, and that too, only as long as it does not introduce a new conversation y in $\mathcal{C}'$ for an action c supported by $\mathcal{C}$ such that y is not a fragment (subset) of a conversation x of c in $\mathcal{C}$ itself.

THEOREM 1. **(Substitutivity of consistency interfaces)** *Let $\mathcal{C}_1$, $\mathcal{C}'_1$, and $\mathcal{C}_2$ be three consistency interfaces such that $comp(\mathcal{C}_1, \mathcal{C}_2)$ and $comp(\mathcal{C}'_1, \mathcal{C}_2)$. Let $\psi = a \not\rightsquigarrow S$ be a specification such that a is supported by either $\mathcal{C}_1$ or $\mathcal{C}_2$. If $(\mathcal{C}_1 \parallel \mathcal{C}_2) \models \psi$ and $\mathcal{C}'_1 \preccurlyeq \mathcal{C}_1$, then $(\mathcal{C}'_1 \parallel \mathcal{C}_2) \models \psi$.*

The above theorem allows a developer to substitute a service $\mathcal{C}'$ in place of an abstract place-holder $\mathcal{C}$ in a design, if $\mathcal{C}'$ refines $\mathcal{C}$. Thus, our framework allows *compositional refinement*: separate parts of a design can be independently refined, say by independent development teams in a parallelized development environment, or even by separate companies that do not want to disclose any information about their code, without having to worry about global consistency issues. Once interfaces are decided at the top level, they can be handed off to separate development teams to be implemented in a more concrete form. As long as the design produced by each team is a refinement of the interface it had been handed to start with, the design of the entire application is guaranteed to be correct. Formally, for consistency interfaces $\mathcal{C}_1$, $\mathcal{C}'_1$, and $\mathcal{C}_2$, if $comp(\mathcal{C}_1, \mathcal{C}_2)$ and $comp(\mathcal{C}'_1, \mathcal{C}_2)$ and $\mathcal{C}'_1 \preccurlyeq \mathcal{C}_1$, then $(\mathcal{C}'_1 \parallel \mathcal{C}_2) \preccurlyeq (\mathcal{C}_1 \parallel \mathcal{C}_2)$. The refinement-checking problem for consistency interfaces is in NP.

EXAMPLE 10. **(Substitutivity of consistency interfaces)** The interface $\mathcal{C}'_{\texttt{Shop}}$ (from Example 9) is a refinement of interface $\mathcal{C}_{\texttt{Shop}}$ (from Example 5), because i) $sig(\mathcal{C}'_{\texttt{Shop}}) = \mathcal{S}'_{\texttt{Shop}}$ is a refinement of $sig(\mathcal{C}_{\texttt{Shop}}) = \mathcal{S}_{\texttt{Shop}}$, and ii) actions supported in $\mathcal{C}'_{\texttt{Shop}}$ require less actions than those supported in $\mathcal{C}_{\texttt{Shop}}$ ($\mathcal{C}'_{\texttt{Shop}}$ does not require the actions for shipping) and $\mathcal{C}'_{\texttt{Shop}}$ supports an additional action $\langle \texttt{SellItem}, \texttt{NOTFOUND}\rangle$, which requires only actions already used by $\mathcal{C}_{\texttt{Shop}}$. Applying Theorem 1 we conclude that if we replace interface $\mathcal{C}_{\texttt{Shop}}$ by its refinement $\mathcal{C}'_{\texttt{Shop}}$ in the composition with the `Store` interface $\mathcal{C}_{\texttt{Store}}$, the new composition is a refinement of the old composition, and the specification ψ from Example 8 is satisfied by the new composition:

$(\mathcal{C}'_{\texttt{Shop}} \parallel \mathcal{C}_{\texttt{Store}}) \preccurlyeq (\mathcal{C}_{\texttt{Shop}} \parallel \mathcal{C}_{\texttt{Store}})$ and $(\mathcal{C}'_{\texttt{Shop}} \parallel \mathcal{C}_{\texttt{Store}}) \models \psi$. ∎

4. PROTOCOL INTERFACES

Consistency interfaces and the associated specification formalism are used to reason about *sets* of actions. Though it is sufficient to catch a large set of web service errors and represent many properties of interest, there is often a need for a richer model of web service behavior that allows reasoning about evolution of behavior over time. In addition to reasoning about *sequences* of actions, it is important to be able to model thread creation, parallel executions of multiple threads, and joining threads after a parallel call. We introduce the formalism of protocol interfaces which is a rich model for representing web service behavior.

The set of *terms* over a set of actions $\mathcal{A}$ is defined by the following grammar ($a, b \in \mathcal{A}$):

$$term \quad :: \quad a \mid a \sqcup b \mid a \sqcap b \mid a \boxplus b$$

A *protocol automaton* is a tuple $\mathcal{F} = (Q, \bot, \delta)$, where Q is a set of control *locations*, $\bot \in Q$ is the *return* location, and $\delta : (Q \setminus \{\bot\}) \to (terms \times Q)$ is the *switch function* of the protocol automaton, which assigns to each location different from $\bot$ a *term* and a successor location. The execution halts when location $\bot$ is reached. A *protocol interface* $\mathcal{P}$ is a pair $(\mathcal{D}, \mathcal{F})$, where $\mathcal{D} : \mathcal{A} \to 2^Q$ is a partial function that assigns to an action a set of locations, and $\mathcal{F}$ is a *protocol automaton*.

The semantics of a protocol interface is presented informally as follows. A formal semantics will be presented in Section 4.2. Intuitively, the execution of an action a starts in one of the locations in $\mathcal{D}(a)$. A switch of the automaton $\delta(q) = (term, q')$ means that, if the automaton is in location q, it recursively invokes *term*, and remembers the successor location q' as the return location, where control returns when the recursive invocation of *term* terminates. If the automaton reaches location $\bot$, it returns control to the return location; a return for the very first invocation of action a leads to termination of execution. The term $a = \langle m, o\rangle$ represents a call to web method m with expected outcome o. The term $a \sqcup b$ represents a nondeterministic choice between a and b. The term $a \sqcap b$ represents spawning two threads for a and b in parallel, while the parent thread waits for both to finish. The term $a \boxplus b$ represents spawning two threads for a and b in parallel, while the parent thread waits for whichever child finishes first. Note that all three of $\sqcup$, $\sqcap$, and $\boxplus$ are commutative, and $\sqcup$ is idempotent while $\sqcap$ and $\boxplus$ are not.

A location q is *well-formed* if there exists a terminating execution of $\mathcal{F}$ starting in location q. An action a is *well-formed* if at least one of the locations in $\mathcal{D}(a)$ is well-formed. To formally define the notion of well-formedness of locations, we use a function $wf : Q \to \{\text{T}, \text{F}\}$, which is inductively defined as follows:

$$wf(\bot) = \text{T}.$$

$$wf(q) = \bigvee_{q_a \in \mathcal{D}(a)} wf(q_a) \ \wedge \ wf(q'), \qquad \text{if } \delta(q) = (a, q').$$

$$wf(q) = \bigvee_{q_a \in \mathcal{D}(a), q_b \in \mathcal{D}(b)} (\ wf(q_a) \wedge wf(q_b)\ ) \ \wedge \ wf(q'), \qquad \text{if } \delta(q) = (a \sqcap b, q').$$

$$wf(q) = \bigvee_{q_a \in \mathcal{D}(a), q_b \in \mathcal{D}(b)} (\ wf(q_a) \vee wf(q_b)\ ) \ \wedge \ wf(q'), \qquad \text{if } \delta(q) = (a \circ b, q'), \circ \in \{\sqcup, \boxplus\}.$$

The function wf can be computed using a least fixed point computation (starting from the function that maps all locations to F) that converges in time $O(n^2 \cdot k^2)$, where $n = |Q|$ is the number of locations of the protocol automaton and $k = \max_a(|\mathcal{D}(a)|)$ is the maximal nondeterministic-branching factor of the interface.

Given a protocol interface $\mathcal{P} = (\mathcal{D}, \mathcal{F})$, the *underlying web service signature* of $\mathcal{P}$ (denoted $sig(\mathcal{P})$) is the partial function $\mathcal{S} : \mathcal{A} \to 2^{\mathcal{A}}$ such that $\mathcal{S}(a) = \bigcup_{q \in \mathcal{D}(a)} sigl(q)$ if $\mathcal{D}(a)$ is defined, and $\mathcal{S}(a)$ is undefined otherwise. The function $sigl : Q \to 2^{\mathcal{A}}$ assigns a set of actions to every location $q \in Q$ of the protocol interface, and is inductively defined as follows: $sigl(q) = g(term) \cup sigl(q')$ for $\delta(q) = (term, q')$ with $q \neq \bot$, and $sigl(\bot) = \emptyset$. The function $g : terms \to 2^{\mathcal{A}}$ is inductively defined as $g(a) = a$, $g(a \circ b) = \{a, b\}$ with $\circ \in \{\sqcup, \sqcap, \boxplus\}$, with $a, b \in \mathcal{A}$. A protocol interface $\mathcal{P}$ is *well-formed*, if $sig(\mathcal{P})$ is well-formed and all actions supported by $\mathcal{P}$ are well-formed. In the rest of the paper, "protocol interface" stands for "well-formed protocol interface".

EXAMPLE 11. **(Well-formed protocol interface)** Let us model the `Shop` web service as a protocol interface. For simplicity, protocol interfaces are defined concisely here by giving the switch function δ of the automaton as a sequence of triples $q : (term, q')$, and the partial function $\mathcal{D}$ is indicated by writing an action in front of every location it is mapped to.

$\mathcal{P}_{\texttt{Shop}} = \{$

$\langle \texttt{SellItem}, \texttt{SOLD}\rangle \mapsto$	$q_0 : (\langle \texttt{ChkAvail}, \texttt{OK}\rangle, q_1)$
	$q_1 : (\langle \texttt{ProcPay}, \texttt{OK}\rangle, \bot)$
$\langle \texttt{SellItem}, \texttt{FAIL}\rangle \mapsto$	$q_2 : (\langle \texttt{ChkAvail}, \texttt{FAIL}\rangle \sqcup \langle \texttt{SellStep1}, \texttt{FAIL}\rangle, \bot)$
$\langle \texttt{SellStep1}, \texttt{FAIL}\rangle \mapsto$	$q_3 : (\langle \texttt{ChkAvail}, \texttt{OK}\rangle, q_4)$
	$q_4 : (\langle \texttt{ProcPay}, \texttt{FAIL}\rangle \sqcup \langle \texttt{SellStep2}, \texttt{FAIL}\rangle, \bot)$
$\langle \texttt{SellStep2}, \texttt{FAIL}\rangle \mapsto$	$q_5 : (\langle \texttt{ProcPay}, \texttt{OK}\rangle, q_6)$
	$q_6 : (\langle \texttt{ShipItem}, \texttt{FAIL}\rangle, \bot)$
$\langle \texttt{ChkAvail}, \texttt{OK}\rangle \mapsto$	$q_7 : (\langle \texttt{ChkStore}, \texttt{OK}\rangle, \bot)$
$\langle \texttt{ChkAvail}, \texttt{FAIL}\rangle \mapsto$	$q_8 : (\langle \texttt{ChkStore}, \texttt{FAIL}\rangle, \bot)$

$\}$

Compare this protocol interface with the consistency interface $\mathcal{C}_{\texttt{Shop}}$; they have a relationship and we will formalize the idea in Section 4.4.

The protocol interface models that for action $\langle \texttt{SellItem}, \texttt{SOLD}\rangle$ the three actions occur in the given sequence in a conversation. When the action $\langle \texttt{SellItem}, \texttt{FAIL}\rangle$ is invoked, `Shop` nondeterministically can check the availability of the item (using `ChkAvail`) with the expectation

of failure, or can invoke `SellStep1` with the expectation of failure. In the latter case, `ChkAvail` is invoked with expectation of success, and then either the payment processing web method (`ProcPay`) or the `Sellstep2` web method is invoked, in both cases with expectation of failure. The invocation of $\langle$`SellStep2`, `FAIL`$\rangle$ leads to a successful invocation of the payment processing web method, and then a failed attempt to ship the item sold (using `ShipItem`).

The protocol interface $\mathcal{P}_{\texttt{Shop}}$ is well-formed, because its underlying signature is well-formed and all its actions are well-formed. ∎

EXAMPLE 12. **(Concurrency and nondeterministic actions)** The following interface is the protocol interface for the `Store` web service:

$$
\begin{aligned}
&\mathcal{P}_{\texttt{Store}} = \{\\
&\ \langle \texttt{ChkStore}, \texttt{OK}\rangle \mapsto \quad \bot\\
&\ \langle \texttt{ChkStore}, \texttt{OK}\rangle \mapsto \quad q_{10}\\
&\ \langle \texttt{ChkStore}, \texttt{FAIL}\rangle \mapsto \quad q_{10} : (\langle \texttt{Supp1.GetOffer}, \texttt{REC}\rangle \sqcap\\
&\qquad\qquad\qquad\qquad \langle \texttt{Supp2.GetOffer}, \texttt{REC}\rangle, q_{11})\\
&\qquad\qquad\qquad\quad q_{11} : (\langle \texttt{Supp1.Order}, \texttt{OK}\rangle \sqcup\\
&\qquad\qquad\qquad\qquad \langle \texttt{Supp2.Order}, \texttt{OK}\rangle, \bot)\\
&\}
\end{aligned}
$$

Let us first consider action $\langle$`ChkStore`, `FAIL`$\rangle$. The interface models for this action that two different supplier web services are simultaneously asked to make an offer. This example shows how the protocol interface expresses not only sequence, but also concurrency. After both offers are received, the automaton switches to location q_{11}, where the `Store` orders the missing item from one of the two suppliers. After this action, the automaton switches to the return location $\bot$, and the conversation induced by action $\langle$`ChkStore`, `FAIL`$\rangle$ terminates.

The invocation of action $\langle$`ChkStore`, `OK`$\rangle$ either immediately returns, or orders new items, when the stock is below a certain threshold. This is modeled by assigning two different locations to the same action. Note that for ordering new items the implementation of $\langle$`ChkStore`, `FAIL`$\rangle$ is shared. ∎

4.1 Compatibility and Composition

Intuitively, as for consistency interfaces, given two protocol interfaces, if their behavior allows them to co-operate, we say they are *compatible*, and we are able to *compose* them, forgetting their internal distinctions and allowing ourselves to focus instead on their behavioral contract with their environment.

Given two protocol interfaces $\mathcal{P}_1 = (\mathcal{D}_1, \mathcal{F}_1)$ and $\mathcal{P}_2 = (\mathcal{D}_2, \mathcal{F}_2)$, if the underlying signatures $sig(\mathcal{P}_1)$ and $sig(\mathcal{P}_2)$ are compatible, and $Q_1 \cap Q_2 = \{\bot\}$, and $\mathcal{P}_c = (\mathcal{D}_c, \mathcal{F}_c)$ is a protocol interface, where $\mathcal{D}_c = \mathcal{D}_1 \cup \mathcal{D}_2$, $\mathcal{F}_c = (Q_1 \cup Q_2, \bot, \delta_1 \cup \delta_2)$, where Q_i and δ_i are the set of locations and the switch function of the automaton $\mathcal{F}_i$ for $i \in \{1, 2\}$, then $\mathcal{P}_1$ and $\mathcal{P}_2$ are *compatible* (denoted $comp(\mathcal{P}_1, \mathcal{P}_2)$), and their *composition* (denoted by $\mathcal{P}_1 \parallel \mathcal{P}_2$) is $\mathcal{P}_c$. The composition operation is commutative and associative. Compatibility and composition of protocol interfaces can be computed in linear time.

A protocol interface $\mathcal{P}$ is *closed* if its underlying web service signature $sig(\mathcal{P})$ supports all actions it requires, formally, $\forall a \in \mathcal{A} : a \in R_{\mathcal{P}} \Rightarrow a \in \text{dom}(sig(\mathcal{P}))$, where $R_{\mathcal{P}} = \{a \mid \exists a' \in \mathcal{A} : a \in sig(\mathcal{P})(a')\}$ is the set of actions required by $\mathcal{P}$. An interface that is not closed is called *open*. Given an open protocol interface $\mathcal{P}$, an *environment* for $\mathcal{P}$ is a protocol interface $\mathcal{E}_{\mathcal{P}}$ that is compatible with $\mathcal{P}$ and supports all actions that are required but not supported by $\mathcal{P}$. Note that the composition $(\mathcal{P} \parallel \mathcal{E}_{\mathcal{P}})$ is closed, and $\mathcal{E}_{\mathcal{P}}$ is not unique. Intuitively, each $\mathcal{E}_{\mathcal{P}}$ represents a design context in which $\mathcal{P}$ can be used.

EXAMPLE 13. **(Compatibility of protocol interfaces)** The two protocol interfaces $\mathcal{P}_{\texttt{Shop}}$ and $\mathcal{P}_{\texttt{Store}}$ are compatible, because their underlying signatures are compatible, their protocol automata have no location in common except $\bot$, and all their actions are well-formed. Thus their composition $\mathcal{P}_{\texttt{Shop}} \parallel \mathcal{P}_{\texttt{Store}}$ is a protocol interface. ∎

4.2 Specifications

We use a specification language which is capable of expressing temporal safety constraints to represent protocol properties. In the context of protocol interfaces, a *conversation* is a set of *sequences* of objects A, where each A is a set of actions; intuitively, it is a directed acyclic graph of actions that are exhibited in sequence or in parallel. Sets of conversations are represented by protocol automata, which concisely represent multi-threaded systems with an unbounded number of threads.

A *specification* ψ for a protocol interface is a formula $a \not\rightsquigarrow \varphi$ where $a \in \mathcal{A}$, and φ is a temporal formula of the form $(\neg C)\ \mathcal{U}\ B$ ("not C until B"), with $C, B \subseteq \mathcal{A}$. Intuitively, a specification $a \not\rightsquigarrow (\neg C)\ \mathcal{U}\ B$ means that the invocation of action a must not lead to a conversation in which an action from the set B occurs before any action from the set C has occurred. The expressions φ can be extended to allow right-associative nesting of $\mathcal{U}$ operators, as well as boolean combinations of $\mathcal{U}$ formulas. We omit such extensions here for simplicity of presentation.

A specification ψ for a protocol interface $\mathcal{P}$ is interpreted over traces generated by the *underlying transition relation* of $\mathcal{P}$, which is defined as follows.

Underlying transition relation. Given a finite set of *symbols* L, a (binary) *tree* t over L is a partial function $t : \mathbb{B}^* \to L$, where $\mathbb{B}^*$ denotes the set of finite words over $\mathbb{B} = \{0, 1\}$, and the domain $\text{dom}(t) = \{p \in \mathbb{B}^* \mid \exists (p, l) \in t\}$ is prefix-closed. Each element from $\text{dom}(t)$ represents a *node* of tree t, and each node p is named with the symbol $t(p)$. The root is represented by the empty word ρ. The set of child nodes of node p in tree t is denoted by $ch(t, p) = \{p' \mid \exists b \in \mathbb{B} : p' = p \cdot b \ \wedge\ p' \in \text{dom}(t)\}$, where $\cdot$ is the concatenation operator. The set of leaf nodes of a tree t is $leaf(t) = \{p \in \text{dom}(t) \mid ch(t, p) = \emptyset\}$. The set of all trees over a finite set L is denoted $\mathcal{T}(L)$.

Given a protocol interface $\mathcal{P} = (\mathcal{D}, \mathcal{F})$, the *underlying transition relation* of $\mathcal{P}$ is a labeled transition relation $\rightarrow_{\mathcal{P}} \subseteq \mathcal{T}(Q^{\boxplus}) \times 2^{\mathcal{A} \cup \{ret\}} \times \mathcal{T}(Q^{\boxplus})$, where the states are trees over the tree symbols $Q^{\boxplus} = Q \times \{\boxplus, \circ\}$, where Q is the set of locations of protocol automaton $\mathcal{F}$, and the transitions between states are labeled with sets of elements from $\mathcal{A} \cup \{ret\}$. We write $t \xrightarrow{A} t'$ for $(t, A, t') \in \rightarrow_{\mathcal{P}}$. In the rules below, if action c is supported by $\mathcal{P}$, q_c is an element of $\mathcal{D}(c)$, otherwise q_c is $\bot$. The relation $\rightarrow_{\mathcal{P}}$ is defined as follows:

- Call: $t \xrightarrow{\{a\}} t'$ if there exists a node p such that $p \in leaf(t)$, $t(p) = q\circ$, and $\delta(q) = (a, q')$ is a switch of $\mathcal{F}$, and $t' = (t \setminus \{(p, q\circ)\}) \cup \{(p, q'\circ), (p \cdot 0, q_a\circ)\}$.

- Fork: $t \xrightarrow{\{a,b\}} t'$ if there exists a node p such that $p \in \mathit{leaf}(t)$, $t(p) = q\circ$, and $\delta(q) = (a \sqcap b, q')$ is a switch of $\mathcal{F}$, and $t' = (t \setminus \{(p, q\circ)\}) \cup \{(p, q'\circ), (p \cdot 0, q_a\circ), (p \cdot 1, q_b\circ)\}$.

- Choice: $t \xrightarrow{\{c\}} t'$ if there exists a node p such that $p \in \mathit{leaf}(t)$, $t(p) = q\circ$, and $\delta(q) = (a \sqcup b, q')$ is a switch of $\mathcal{F}$, and $t' = (t \setminus \{(p, q\circ)\}) \cup \{(p, q'\circ), (p \cdot 0, q_c\circ)\}$, where $c \in \{a, b\}$.

- Fork-Choice: $t \xrightarrow{\{a,b\}} t'$ if there exists a node p such that $p \in \mathit{leaf}(t)$, $t(p) = q\circ$, and $\delta(q) = (a \boxplus b, q')$ is a switch of $\mathcal{F}$, and $t' = (t \setminus \{(p, q\circ)\}) \cup \{(p, q'\boxplus), (p \cdot 0, q_a\circ), (p \cdot 1, q_b\circ)\}$.

- Return: $t \xrightarrow{\{ret\}} t'$ if there exists a node $p \cdot b$, where $b \in \mathbb{B}$, such that $p \cdot b \in \mathit{leaf}(t)$, $t(p \cdot b) = \bot\circ$, $t(p) = q\circ$, and $t' = t \setminus \{(p \cdot b, \bot\circ)\}$.

- Return & Remove Sibling Tree: $t \xrightarrow{\{ret\}} t'$ if there exists a node $p \cdot b$, $b \in \mathbb{B}$ such that $p \cdot b \in \mathit{leaf}(t)$, $t(p \cdot b) = \bot\circ$, $t(p) = q\boxplus$, and $t' = (t \setminus \{(p \cdot p', q'?) \mid p' \in \mathbb{B}^* \wedge q'? \in Q^{\boxplus}\}) \cup \{(p, q\circ)\}$.

Intuitively, we have a pushdown system whose state is a tree, not a stack (tree with branching out-degree 1). The primitives Call, Choice, and Return contribute to pushdown behavior, and the primitives Fork and Fork-Choice lead to branching in the pushdown state. The leaves of the tree represent parallel threads of control. A node p of the tree is labeled with an element of $Q^{\boxplus}$, the label representing the "return location" where control should reach when all the children of node p are removed from the tree. For a Call of a supported action, the current location is popped as a leaf from the tree, the successor location is first pushed, and then one of the locations corresponding to the called action (nondeterministically chosen) is pushed as child of the successor location. Choice behaves similarly except that it produces the successor tree for any of the two actions. In case of a Fork and a Fork-Choice, the current location is popped, the successor location is pushed, and a branch point is created in the state, two locations corresponding to the two parallel actions are pushed as children of the successor location. In case of Return, a leaf labeled with the return location $\bot$ is popped. If the parent of the return location is the successor of a fork-choice term, the whole sibling subtree is also removed. In other words, a node with two children resulting from a fork can only be popped if both children were popped before, but a node with two children resulting from a fork-choice is popped if one of the children was popped. To remember this fact in the tree symbols, every control location is paired with a flag from $\{\boxplus, \circ\}$. Note that invocations of unsupported actions are assumed to immediately return.

A *run* of a transition relation is an alternating sequence of trees and sets of actions $t_0, A_1, t_1, A_2, t_2, \ldots$, with $\forall i \in \{1, \ldots n\} : t_{i-1} \xrightarrow{A_i} t_i$. A *trace* is the projection of a run to its action sets, e.g., for the run $t_0, A_1, t_1, A_2, t_2, \ldots$, the corresponding trace is $A_1, A_2, \ldots$; for a location q, a *q-run* is a run $t_0, A_1, t_1, \ldots$ with $t_0 = \{(\rho, q\circ)\}$, i.e., a run starting from location q; a *q-trace* is a trace corresponding to a q-run.

Specification checking. A location q *satisfies* a temporal formula $(\neg C)\ \mathcal{U}\ B$ (written $q \models (\neg C)\ \mathcal{U}\ B$) if there exists a q-trace $A_1, A_2, \ldots$ such that there exists a j such that $A_j \cap B \neq \emptyset$, and for all $i < j$, we have $A_i \cap C = \emptyset$. A closed protocol interface $\mathcal{P} = (\mathcal{D}, \mathcal{F})$ *satisfies* a specification $\psi = a \not\rightsquigarrow \varphi$ (written $\mathcal{P} \models \psi$) if for all $q \in \mathcal{D}(a)$, we have $q \not\models \varphi$.

Although we defined the semantics of $q \models \varphi$ in terms of traces of the underlying transition relation, to check satisfiability in practice we do not compute the transition relation because it may be infinite. Instead, we define a set of proof rules (cf. Figure 2) and give a polynomial-time algorithm (cf. Algorithm 1). Using the proof rules in Figure 2 we inductively decide *judgements* $\Gamma = q \models \varphi$ where φ is a temporal formula of the form $\varphi^{\mathcal{U}} = (\neg C)\ \mathcal{U}\ B$, or $\varphi^{\Box} = \Box(\neg C)$. All leaves of the proof result from the proof rules "Return $\Box$" and "Reached $\mathcal{U}^0$". If a judgement is true, there exists a short proof of linear size, and its existence can be decided in polynomial time. We explain some of the proof rules below.

The rule "Return $\Box$" at the top of Figure 2 asserts that the location $\bot$ satisfies the property $\Box(\neg C)$ i.e., no $\bot$-trace should contain an element from C; which is true, since the $\bot$-trace is empty. The "Call $\Box$" rule says that if the invocation of action a does not (recursively) lead to an element from C, and a itself is not in C, and furthermore, the successor location q', where control returns after execution of a, does not lead to an element from C, then the location q, at which a is called, never leads to an element of C. The rule "Choice $\Box$" for the operator $\sqcup$ is similar to "Call $\Box$", except that one of the nondeterministically chosen actions from the term invoked must fulfill the requirements in the antecedent. The remaining rules similarly enforce the semantics of a recursive system with nondeterministic choice, process-spawning and joining, and a pushdown tree store.

PROPOSITION 1. **(Correctness of specification checking)** *For a given closed protocol interface $\mathcal{P}$ and a specification $\psi = a \not\rightsquigarrow (\neg C)\ \mathcal{U}\ B$, procedure* CheckSpec$(\mathcal{P}, a, B, C)$ *in Algorithm 1 stops with answer* YES *if $\mathcal{P}$ satisfies ψ, and* NO *otherwise.*

Algorithm 1 CheckSpec$(\mathcal{P}, a, B, C)$

Input: Closed protocol interface $\mathcal{P} = (\mathcal{D}, \mathcal{F})$,
Action sets $B, C \subseteq \mathcal{A}$, and action $a \in \mathcal{A}$
Output: YES if $\mathcal{P}$ satisfies $a \not\rightsquigarrow (\neg C)\ \mathcal{U}\ B$, NO otherwise
Variables: Set of judgements S, boolean `done`

```
 1: done := F
 2: while (¬ done) do
 3:    done := T
 4:    for each location q of automaton F do
 5:       // Try to prove q ⊨ □(¬C).
 6:       if all premises of a rule for q ⊨ □(¬C) are in S
          then
 7:          S := S ∪ {q ⊨ □(¬C)}
 8:          done := F
 9:       // Try to prove q ⊨ (¬C) U B.
10:       if all premises of a rule for q ⊨ (¬C) U B are in S
          then
11:          S := S ∪ {q ⊨ (¬C) U B}
12:          done := F
      end
13:   if (q ⊨ (¬C) U B) ∈ S for some q ∈ D(a) then
14:      return NO
    end
15: return YES
```

$$\frac{}{\bot \models \Box(\neg C)} \qquad \text{(Return } \Box\text{)}$$

$$\frac{q_a \models \Box(\neg C) \quad a \notin C \quad q' \models \Box(\neg C)}{q \models \Box(\neg C)} \quad \begin{array}{l}\delta(q) = (a, q'),\\ q_a \in \mathcal{D}(a)\end{array} \qquad \text{(Call } \Box\text{)}$$

$$\frac{q_c \models \Box(\neg C) \quad c \notin C \quad q' \models \Box(\neg C)}{q \models \Box(\neg C)} \quad \begin{array}{l}\delta(q) = (a \sqcup b, q'), c \in \{a, b\},\\ q_c \in \mathcal{D}(c)\end{array} \qquad \text{(Choice } \Box\text{)}$$

$$\frac{q_a \models \Box(\neg C) \quad a \notin C \quad q_b \models \Box(\neg C) \quad b \notin C \quad q' \models \Box(\neg C)}{q \models \Box(\neg C)} \quad \begin{array}{l}\delta(q) = (a \sqcap b, q'),\\ q_a \in \mathcal{D}(a), q_b \in \mathcal{D}(b)\end{array} \qquad \text{(Fork } \Box\text{)}$$

$$\frac{q_c \models \Box(\neg C) \quad a \notin C \quad b \notin C \quad q' \models \Box(\neg C)}{q \models \Box(\neg C)} \quad \begin{array}{l}\delta(q) = (a \boxplus b, q'), c \in \{a, b\},\\ q_c \in \mathcal{D}(c)\end{array} \qquad \text{(Fork-Choice } \Box\text{)}$$

$$\frac{c \in B}{q \models (\neg C)\ \mathcal{U}\ B} \quad \begin{array}{l}\delta(q) = (c, q') \ \vee\ (\ \delta(q) = (a \circ b, q') \ \wedge\ c \in \{a, b\}\ ),\\ q_c \in \mathcal{D}(c), \circ \in \{\sqcup, \sqcap, \boxplus\}\end{array} \qquad \text{(Reached } \mathcal{U}^0\text{)}$$

$$\frac{c \models (\neg C)\ \mathcal{U}\ B \quad c \notin C}{q \models (\neg C)\ \mathcal{U}\ B} \quad \begin{array}{l}\delta(q) = (c, q') \ \vee\ (\ \delta(q) = (a \circ b, q') \ \wedge\ c \in \{a, b\}\ ),\\ q_c \in \mathcal{D}(c), \circ \in \{\sqcup, \sqcap, \boxplus\}\end{array} \qquad \text{(Reached } \mathcal{U}^+\text{)}$$

$$\frac{q_a \models \Box(\neg C) \quad a \notin C \quad q' \models (\neg C)\ \mathcal{U}\ B}{q \models (\neg C)\ \mathcal{U}\ B} \quad \begin{array}{l}\delta(q) = (a, q'),\\ q_a \in \mathcal{D}(a)\end{array} \qquad \text{(Call } \mathcal{U}\text{)}$$

$$\frac{q_c \models \Box(\neg C) \quad c \notin C \quad q' \models (\neg C)\ \mathcal{U}\ B}{q \models (\neg C)\ \mathcal{U}\ B} \quad \begin{array}{l}\delta(q) = (a \sqcup b, q'), c \in \{a, b\},\\ q_c \in \mathcal{D}(c)\end{array} \qquad \text{(Choice } \mathcal{U}\text{)}$$

$$\frac{q_a \models \Box(\neg C) \quad a \notin C \quad q_b \models \Box(\neg C) \quad b \notin C \quad q' \models (\neg C)\ \mathcal{U}\ B}{q \models (\neg C)\ \mathcal{U}\ B} \quad \begin{array}{l}\delta(q) = (a \sqcap b, q'),\\ q_a \in \mathcal{D}(a), q_b \in \mathcal{D}(b)\end{array} \qquad \text{(Fork } \mathcal{U}\text{)}$$

$$\frac{q_c \models \Box(\neg C) \quad a \notin C \quad b \notin C \quad q' \models (\neg C)\ \mathcal{U}\ B}{q \models (\neg C)\ \mathcal{U}\ B} \quad \begin{array}{l}\delta(q) = (a \boxplus b, q'), c \in \{a, b\},\\ q_c \in \mathcal{D}(c)\end{array} \qquad \text{(Fork-Choice } \mathcal{U}\text{)}$$

Figure 2: Proof rules for specification checking

The algorithm computes the least fixed point of the set of provable judgements w.r.t. the proof rules by starting with the empty set of judgements, and trying to prove every still-unproven judgement until no new judgements can be proved. For every location q, it iterates through the all rules that can be used to prove the two judgements $q \models (\neg C)\ \mathcal{U}\ B$ and $q \models \Box(\neg C)$. If all antecedents of a rule are already proven, the rule is "triggered", and the consequent is considered proven. This corresponds to a leaves-to-root traversal of the proof for each judgement, where a proof for a judgement Γ is seen as a tree with Γ as root, instances of the proof rules as internal nodes, and judgements axiomatically known to be true, as leaves. Since there are only finitely many proof rules for each location, finitely many locations, and finitely many judgements, and the algorithm proves at least one new judgement in each iteration till the fixed point is reached, the procedure must terminate. For each judgement Γ for which a proof exists, a shortest proof $p(\Gamma)$ exists. By induction over the length of $p(\Gamma)$, we conclude that every Γ for which a proof exists, is eventually proved by our algorithm, and vice versa.

PROPOSITION 2. **(Complexity of specification checking)** *Given a closed protocol interface $\mathcal{P} = (\mathcal{D}, \mathcal{F})$, and given a specification $\psi = a \not\rightsquigarrow (\neg C)\ \mathcal{U}\ B$, the question if $\mathcal{P}$ satisfies ψ can be decided in $O(n^2 \cdot k^2)$ time, where n is the number of locations of $\mathcal{F}$, and $k = \max_{a \in \mathcal{A}} |\mathcal{D}(a)|$ is the maximal nondeterministic-branching factor of $\mathcal{P}$.*

The set operations insertion and membership-checking can be done in $O(1)$ time, using, for example, a direct-access array of flags, since the number of judgements is bounded. A term in a switch can have the form $a \sqcap b$; hence the algorithm, in the worst case, has to check $O(k^2)$ proof rules for a location (lines 6 and 10). Line 13 takes $O(k)$ time. Therefore, lines 5 to 14 altogether are in $O(k^2)$ time. The loop over locations (line 4) always makes n iterations. The fixpoint iteration (line 2) is executed at most $2n$ times since we have only $2n$ judgements. Note that for the special case when C is empty, the problem reduces to reachability of a state where an action from B is invoked, which can be decided in linear time.

EXAMPLE 14. **(Specification for protocol interfaces)** The `Shop` web service is not allowed to process payments if the item to be sold is not available. Therefore, we need to check that in $\mathcal{P}_{\texttt{Shop}}$ the payment is never processed before the availability of the item is checked; this property is expressed in the following specification ψ:

$$\langle \texttt{SellItem}, \texttt{FAIL} \rangle \not\rightarrow \ \neg\{\langle \texttt{ChkStore}, \texttt{OK} \rangle\} \,\mathcal{U}\, \{\langle \texttt{ProcPay}, \texttt{OK} \rangle\}$$

This property is satisfied if *none* of the conversations of $\mathcal{P}_{\texttt{Shop}}$ has the following characteristic: it starts with a sequence of actions which are all different from $\langle \texttt{ChkStore}, \texttt{OK} \rangle$, and eventually an action $\langle \texttt{ProcPay}, \texttt{OK} \rangle$ occurs. Note that this particular specification is "stronger" than the corresponding specification we discussed for consistency interfaces: it forbids not only the conversations that *fail* checking of availability before payment, but also those that *do not check* for availability at all; and also the desired sequence is enforced. Observe that the composition $\mathcal{P}_c = \mathcal{P}_{\texttt{Shop}} \parallel \mathcal{P}_{\texttt{Store}}$, along with a *minimal* environment for $\mathcal{P}_c$ that does not require any actions, satisfies the specification ψ.

Properties like ψ in general cannot be verified using consistency interfaces, because consistency interfaces do not keep track on the order of actions in conversations. However, a subset of the protocol specifications can be checked even using consistency interfaces using a relationship between consistency and protocol interfaces (intuitively "conservative extension") that will be formalized in Section 4.4. ∎

4.3 Substitutivity

As in the case of consistency interfaces, we want to enable the substitution of a protocol interface in a design comprising a set of web services, with the assurance that the change will not allow violation by the combined system of any specification that it satisfied before the substitution. Our notion of refinement for protocol interfaces is based on simulation of labeled transition systems. We first define the notions labeled transition system and simulation. Then we describe how to build a labeled transition system from the underlying transition relation of the protocol interface, and define refinement of two protocol interfaces based on their labeled transition systems.

A labeled transition system (LTS) $\mathcal{TS}$ is a tuple $(S, S_i, L, \rightarrow)$, where S the set of *states*, $S_i \subseteq S$ is the set of *initial states*, L is the set of *labels*, and $\rightarrow \subseteq S \times L \times S$ is the transition relation. An LTS $\mathcal{TS}' = (S', S'_i, L, \rightarrow')$ *is simulated by* an LTS $\mathcal{TS} = (S, S_i, L, \rightarrow)$ if there exists a relation $\precsim \subseteq S' \times S$ such that:

- for every $s_1 \in S$, $s'_1 \in S'$, if $s'_1 \precsim s_1$, then for every transition $s'_1 \xrightarrow{l} s'_2$, there exists a transition $s_1 \xrightarrow{l} s_2$, such that $s'_2 \precsim s_2$, and
- for every $s' \in S'_i$, there exists $s \in S_i$ such that $s' \precsim s$.

Given a protocol interface $\mathcal{P} = (\mathcal{D}, \mathcal{F})$ and an action a supported by $\mathcal{P}$, the *underlying transition system obtained by invoking a on* $\mathcal{P}$ (denoted $uts(\mathcal{P}, a)$) is the LTS $\mathcal{TS} = (\mathcal{T}(Q^{\boxplus}), \{\{\rho \mapsto q\circ\} \mid q \in \mathcal{D}(a)\}, 2^{\mathcal{A} \cup \{ret\}}, \rightarrow_{\mathcal{P}})$, where $\mathcal{T}(Q^{\boxplus})$ is the set of trees over the set of labels $Q^{\boxplus}$, and $Q^{\boxplus} = Q \times \{\boxplus, \circ\}$, where Q is the set of locations in $\mathcal{F}$, and $\rightarrow_{\mathcal{P}}$ is the underlying transition relation of $\mathcal{P}$ defined in Section 4.2. Intuitively, it is an LTS with a set of states comprising the set of binary trees with nodes labeled with elements from $Q^{\boxplus}$, the set of initial states comprising trees with a root labeled with an element of $\mathcal{D}(a)$, and the transition relation $\rightarrow_{\mathcal{P}}$.

Given two protocol interfaces $\mathcal{P}$ and $\mathcal{P}'$, we say $\mathcal{P}'$ *refines* $\mathcal{P}$ (written $\mathcal{P}' \preccurlyeq \mathcal{P}$), if

- $sig(\mathcal{P}') \preccurlyeq sig(\mathcal{P})$, and
- for every action $a \in \mathcal{A}$, if $\mathcal{P}$ supports a, then the LTS $\mathcal{TS}' = uts(\mathcal{P}', a)$ is simulated by the LTS $\mathcal{TS} = uts(\mathcal{P}, a)$.

Note that for all protocol interfaces $\mathcal{P}$ and $\mathcal{P}'$, we have $\mathcal{P}' \preccurlyeq \mathcal{P}$ if and only if $(\mathcal{P}' \parallel \mathcal{E}^0_{\mathcal{P}}) \preccurlyeq (\mathcal{P} \parallel \mathcal{E}^0_{\mathcal{P}})$, where $\mathcal{E}^0_{\mathcal{P}}$ is the *minimal* environment of $\mathcal{P}$ and $\mathcal{P}'$, which supports each action required but not supported by $\mathcal{P}$ and $\mathcal{P}'$ with an implementation that returns immediately on invocation.

PROPOSITION 3. **(Compositionality of refinement)** *Let $\mathcal{P}_1$, $\mathcal{P}'_1$, and $\mathcal{P}_2$ be three protocol interfaces such that comp$(\mathcal{P}_1, \mathcal{P}_2)$ and comp$(\mathcal{P}'_1, \mathcal{P}_2)$. If $\mathcal{P}'_1 \preccurlyeq \mathcal{P}_1$, then $(\mathcal{P}'_1 \parallel \mathcal{P}_2) \preccurlyeq (\mathcal{P}_1 \parallel \mathcal{P}_2)$.*

From this proposition it follows that a component-based system, after refinement of one component, still satisfies all specifications that it satisfied before refinement.

THEOREM 2. **(Substitutivity of protocol interfaces)** *Let $\mathcal{P}_1$, $\mathcal{P}'_1$, and $\mathcal{P}_2$ be three protocol interfaces such that comp$(\mathcal{P}_1, \mathcal{P}_2)$ and comp$(\mathcal{P}'_1, \mathcal{P}_2)$; and $(\mathcal{P}_1 \parallel \mathcal{P}_2)$ and $(\mathcal{P}'_1 \parallel \mathcal{P}_2)$ are both closed. Let $\psi = a \not\rightarrow (\neg C)\,\mathcal{U}\, B$ be a specification such that a is supported by $\mathcal{P}_1$. Then, if $(\mathcal{P}_1 \parallel \mathcal{P}_2) \models \psi$ and $\mathcal{P}'_1 \preccurlyeq \mathcal{P}_1$, then $(\mathcal{P}'_1 \parallel \mathcal{P}_2) \models \psi$.*

Note that given protocol interfaces $\mathcal{P} = (\mathcal{D}, \mathcal{F})$ and $\mathcal{P}' = (\mathcal{D}', \mathcal{F}')$, deciding the question if $\mathcal{P}' \preccurlyeq \mathcal{P}$ requires, according to the definition of refinement above, checking whether the corresponding underlying transition systems are in simulation. However, the underlying transition systems may be infinite-state systems: there can be an infinite number of trees representing the configuration of the parallel and sequential action invocations in the multi-threaded system. Our formalism of protocol automata and the underlying transition relation enables visibility of recursion and control returns [1], therefore, the problem can be solved efficiently as follows. We define a *local simulation* relation on locations: $\lhd \subseteq Q' \times Q$, where Q, Q' are the sets of locations of $\mathcal{F}$ and $\mathcal{F}'$, as follows: $(q', q) \in \lhd$ if the LTS pair $\mathcal{TS}' = (\mathcal{T}(Q'^{\boxplus}), \{\{\rho \mapsto q'\circ\}\}, 2^{\mathcal{A} \cup \{ret\}}, \rightarrow_{\mathcal{P}'})$ and $\mathcal{TS} = (\mathcal{T}(Q^{\boxplus}), \{\{\rho \mapsto q\circ\}\}, 2^{\mathcal{A} \cup \{ret\}}, \rightarrow_{\mathcal{P}})$ are such that $\mathcal{TS}' \precsim \mathcal{TS}$. Note that we can compute the relation $\lhd$ using a greatest-fixed point computation starting from the set of all pairs $\{(q', q) \mid q'$ location of $\mathcal{F}', q$ location of $\mathcal{F}\}$ and removing pairs (q', q) when the one-step simulation locally fails, i.e., q has no switches to match a term that q' can invoke; until a fixed point is reached and for every action a supported by $\mathcal{P}$, for every location $q' \in \mathcal{D}'(a)$ there is a location $q \in \mathcal{D}(a)$ such that the pair (q', q) is in the fixed point set, and we conclude simulation holds; or there exists at least one action a supported by $\mathcal{P}$ such that all the pairs $\{(q', q) \mid q' \in \mathcal{D}'(a), q \in \mathcal{D}(a)\}$ are eventually removed from the set of pairs, at which point we conclude that simulation fails at some point on invocation of action a. Note that the other condition for refinement involves checking refinement on the underlying signatures $sig(\mathcal{P})$ and $sig(\mathcal{P}')$, which takes linear time.

PROPOSITION 4. **(Complexity of refinement checking)** *Given protocol interfaces $\mathcal{P} = (\mathcal{D}, \mathcal{F})$ and $\mathcal{P}' = (\mathcal{D}', \mathcal{F}')$, the question if $\mathcal{P}' \preccurlyeq \mathcal{P}$ can be decided in $O(c \cdot (|\mathcal{A}| + c))$ time, where $\mathcal{A}$ is the set of actions, $c = n \cdot n' \cdot k \cdot k'$, where n and n' are the numbers of locations in $\mathcal{F}$ and $\mathcal{F}'$ respectively, $k = \max_{a \in \mathcal{A}}(|\mathcal{D}(a)|)$ and $k' = \max_{a \in \mathcal{A}}(|\mathcal{D}'(a)|)$.*

EXAMPLE 15. **(Substitutivity of protocol interfaces)** Consider the protocol interface $\mathcal{P}_{\texttt{Shop}}$ and the following protocol interface $\mathcal{P}'_{\texttt{Shop}}$:

$\mathcal{P}'_{\texttt{Shop}} = \{$
$\langle \texttt{SellItem}, \texttt{SOLD} \rangle \mapsto \quad q_{20} : (\langle \texttt{ChkAvail}, \texttt{OK} \rangle, q_{21})$
$\qquad\qquad\qquad\qquad\quad q_{21} : (\langle \texttt{ProcPay}, \texttt{OK} \rangle, \bot)$
$\langle \texttt{SellItem}, \texttt{FAIL} \rangle \mapsto \quad q_{22} : (\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \sqcup$
$\qquad\qquad\qquad\qquad\quad \langle \texttt{SellStep1}, \texttt{FAIL} \rangle, \bot)$
$\langle \texttt{SellStep1}, \texttt{FAIL} \rangle \mapsto \quad q_{23} : (\langle \texttt{ChkAvail}, \texttt{OK} \rangle, q_{24})$
$\qquad\qquad\qquad\qquad\quad q_{24} : (\langle \texttt{ProcPay}, \texttt{FAIL} \rangle, \bot)$
$\langle \texttt{SellStep2}, \texttt{FAIL} \rangle \mapsto \quad q_{25} : (\langle \texttt{ProcPay}, \texttt{OK} \rangle, q_{26})$
$\qquad\qquad\qquad\qquad\quad q_{26} : (\langle \texttt{ShipItem}, \texttt{FAIL} \rangle, \bot)$
$\langle \texttt{SellItem}, \texttt{NOTFOUND} \rangle \mapsto \quad q_{27} : (\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle, \bot)$
$\langle \texttt{ChkAvail}, \texttt{OK} \rangle \mapsto \quad q_{28} : (\langle \texttt{ChkStore}, \texttt{OK} \rangle, \bot)$
$\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \mapsto \quad q_{29} : (\langle \texttt{ChkStore}, \texttt{FAIL} \rangle, \bot)$
$\}$

The protocol interface $\mathcal{P}'_{\texttt{Shop}}$ refines interface $\mathcal{P}_{\texttt{Shop}}$ because the refined version "implements" the abstraction: it adds guarantees (it supports $\langle \texttt{SellItem}, \texttt{NOTFOUND} \rangle$), and removes some nondeterministic choice in the behavior (in $\langle \texttt{SellStep1}, \texttt{FAIL} \rangle$). Note that the abstraction nondeterministically generates some traces that cannot be matched by the refinement. ∎

4.4 Consistency and protocol interfaces

We have presented in Sections 3 and 4 two different theories for modeling web service interfaces, with different definitions for composition and refinement. In the following discussion we show that the two theories are consistent with each other, and that the protocol theory is a conservative extension of the consistency theory.

Given a protocol interface $\mathcal{P} = (\mathcal{D}, \mathcal{F})$, the *underlying consistency interface* of $\mathcal{P}$ (denoted $uci(\mathcal{P})$) is the partial function $\mathcal{C} : \mathcal{A} \to \mathcal{B}(\mathcal{A})$ such that $\mathcal{C}(a) = \bigsqcup_{q \in \mathcal{D}(a)} uce(q)$ if $\mathcal{D}(a)$ is defined, and $\mathcal{C}(a)$ is undefined otherwise. The function $uce : Q \to \mathcal{B}(\mathcal{A})$ assigns an expression to every location $q \in Q$ of the protocol interface, and is inductively defined as follows: $uce(q) = f(term) \sqcap uce(q')$ for $\delta(q) = (term, q')$ with $q \neq \bot$, and $uce(\bot) = \top$. The function $f : terms \to \mathcal{B}(\mathcal{A})$ is inductively defined as $f(a) = a$, $f(a \sqcup b) = a \sqcup b$, and $f(a \circ b) = a \sqcap b$ with $\circ \in \{\sqcap, \boxplus\}$, and $a, b \in \mathcal{A}$. Note that $sig(uci(\mathcal{P})) = sig(\mathcal{P})$, as expected, and hence for two protocol interfaces $\mathcal{P}_1$ and $\mathcal{P}_2$, $comp(\mathcal{P}_1, \mathcal{P}_2) \Rightarrow comp(uci(\mathcal{P}_1), uci(\mathcal{P}_2))$, but the converse is not true. Given a protocol specification $\psi_p = a \not\rightsquigarrow (\neg C)\ \mathcal{U}\ B$, the set of *underlying consistency specifications* of ψ_p is defined as $ucs(\psi_p) = \{a \not\rightsquigarrow \{b\} \mid b \in B\}$.

THEOREM 3. **(Conservative extension)** *The theory of protocol interfaces is a conservative extension of the theory of consistency interfaces. In particular:*

1. *Given two compatible protocol interfaces $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $\mathcal{P}_1 \parallel \mathcal{P}_2$ is closed, and a protocol specification ψ_p, if $uci(\mathcal{P}_1) \parallel uci(\mathcal{P}_2) \models \psi_c$ for every $\psi_c \in ucs(\psi_p)$, then $\mathcal{P}_1 \parallel \mathcal{P}_2 \models \psi_p$. The converse is not true.*
2. *Given protocol interfaces $\mathcal{P}$ and $\mathcal{P}'$, if $\mathcal{P}' \preccurlyeq \mathcal{P}$, then $uci(\mathcal{P}') \preccurlyeq uci(\mathcal{P})$. The converse is not true.*

EXAMPLE 16. **(Conservative extension)** Consider the two compatible protocol interfaces $\mathcal{P}_{\texttt{Shop}}$ and $\mathcal{P}_{\texttt{Store}}$ from the previous examples in this section, and the following specification ψ_p:

$\langle \texttt{SellItem}, \texttt{SOLD} \rangle \not\rightsquigarrow \neg\{\}\ \mathcal{U}\ \{\langle \texttt{ProcPay}, \texttt{FAIL} \rangle\}$

This specification requires that a conversation starting with $\langle \texttt{SellItem}, \texttt{SOLD} \rangle$ must not reach the action $\langle \texttt{ProcPay}, \texttt{FAIL} \rangle$, no matter what other actions occur on the way to it. If we apply the definition of underlying consistency interface to the protocol interfaces $\mathcal{P}_{\texttt{Shop}}$ and $\mathcal{P}_{\texttt{Store}}$, we get the underlying consistency interfaces $uci(\mathcal{P}_{\texttt{Shop}})$ and $uci(\mathcal{P}_{\texttt{Store}})$ (which are similar to, but not identical to the examples in Section 3).

The set of underlying consistency specifications for ψ_p is a singleton, it contains the following specification $\psi_c = ucs(\psi_p)$:

$\langle \texttt{SellItem}, \texttt{SOLD} \rangle \not\rightsquigarrow \{\langle \texttt{ProcPay}, \texttt{FAIL} \rangle\}$

Theorem 3 states in its first part that if the composition $uci(\mathcal{P}_{\texttt{Shop}}) \parallel uci(\mathcal{P}_{\texttt{Store}})$ satisfies all of the underlying consistency specifications — which is true in our case — then the composition $\mathcal{P}_{\texttt{Shop}} \parallel \mathcal{P}_{\texttt{Store}}$ satisfies the protocol specification ψ_p. This particular ψ_p is a reachability specification, but in general given a protocol specification that rules out a sequence of actions, the conjunction of the underlying consistency specifications rules out all possible orders of the actions from the forbidden sequence. Then, for a protocol specification ψ_p, the conjunction $\bigwedge ucs(\psi_p)$ of the underlying consistency specifications is "stronger" than ψ_p, which forbids a conversation only if the actions occur in one particular order. Thus, the ability to reason about sequences gives the designer the ability to much more precisely express what kinds of behavior are undesired. ∎

EXAMPLE 17. **(Conservative extension of refinement)** Consider the protocol interfaces $\mathcal{P}_{\texttt{Shop}}$ and $\mathcal{P}'_{\texttt{Shop}}$ from the previous examples. We already know that $\mathcal{P}'_{\texttt{Shop}}$ refines $\mathcal{P}_{\texttt{Shop}}$. Theorem 3 states in its second part that in this case the underlying consistency interfaces fulfill the refinement relation as well. The underlying consistency interfaces of $\mathcal{P}_{\texttt{Shop}}$ and $\mathcal{P}'_{\texttt{Shop}}$ are $uci(\mathcal{P}_{\texttt{Shop}})$ and $uci(\mathcal{P}'_{\texttt{Shop}})$, respectively. The refinement relation $uci(\mathcal{P}'_{\texttt{Shop}}) \preccurlyeq uci(\mathcal{P}_{\texttt{Shop}})$ holds. Note that the underlying consistency interfaces are very similar, but not identical to the examples in Section 3. ∎

5. RELATED WORK

The field of modeling web services and their interaction protocols has received considerable attention in recent years. We will briefly mention some of the related work.

A variety of formalisms have been proposed for e-business conversation models [17], models for control flow [24, 16], and to model dynamic contracts between services [6]. A high-level programming language for web service implementation, XL [10], supports constructs like nondeterministic choice and parallel execution. BPEL[1], together with WS-C[2] and WS-T[3], have emerged as the technology for specifying

[1] `http://www.ibm.com/developerworks/library/ws-bpel/`
[2] `http://xml.coverpages.org/WS-Coordination200309.pdf`
[3] `http://xml.coverpages.org/WS-Transaction2002.pdf`

interaction protocols between web services. They are built on top of the XML-based messaging protocol SOAP[4] and the interface description language WSDL[5].

Verification of BPEL descriptions using finite-state model extraction and the NuSMV model-checker has been proposed [21]. Temporal logics for compositional reasoning about web service interfaces have been proposed [22]. Fu, Bultan, and Su studied conversations which occur in compositions of web services [4, 12], and analyzed interactions of composed web services by modeling them as conversations [13, 14]. An implementation supporting a guarded automaton language and using the SPIN model-checker is available in WSAT [15]. A similar approach was chosen in [19], translating BPEL descriptions of web services into Promela, and using the SPIN model-checker to analyze web service flow. Unanticipated behaviors were detected in the WS-AT[6] protocol using a temporal specification language and model checking [18]. Process interactions have been verified using finite-state representations for web service composition [11].

A DAML-S[7]-based semantics has been proposed for web services [20]. First-order logic is used to express specifications, and web service descriptions are represented as Petri nets for simulation, verification, and composition. A CCS-based formalism to represent and analyze WSCI descriptions has been proposed [3], and the issues of compatibility [23] and substitutivity are discussed.

The protocol interface formalism proposed in this paper supports programming and modeling language constructs like spawning threads, nondeterministic choice, sequencing, recursion, join operations, callbacks, etc., which are supported by web service programming or modeling frameworks like the .NET framework, or WSCI[8]. However, while Turing-complete programming environments or modeling languages suffer from the undecidability of non-trivial properties, temporal specifications and refinement of our interfaces can be algorithmically checked.

Acknowledgement. We thank Nir Piterman for helpful discussions about algorithms for checking properties of pushdown systems.

6. REFERENCES

[1] R. Alur and P. Madhusudan. Visibly pushdown languages. *Proc. STOC*, pp. 202–211. ACM, 2004.

[2] S. Anderson, M. Chapman, M. Goodner, P. Kackinaw, and R. Rekasius. Supply chain management use case model. Working group interim report, Web Services-Interoperability Organization, 2002.

[3] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web service choreographies. *Proc. WS-FM*. Elsevier, 2004.

[4] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. *Proc. WWW*, pp. 403–410. ACM, 2003.

[4] `http://www.w3.org/TR/soap12/`

[5] `http://www.w3.org/TR/wsdl`

[6] `http://www-106.ibm.com/developerworks/library/ws-atomtran/`

[7] `http://www.daml.org/services/`

[8] `http://www.w3c.org/TR/wsci/`

[5] A. Chakrabarti, L. de Alfaro, T. Henzinger, and M. Stoelinga. Resource interfaces. *Proc. EMSOFT*, LNCS 2855, pp. 117–133. Springer, 2003.

[6] H. Davulcu, M. Kifer, and I. V. Ramakrishnan. CTR-S: a logic for specifying contracts in semantic web services. *Proc. WWW*, pp. 144–153. ACM, 2004.

[7] L. de Alfaro and T. Henzinger. Interface automata. *Proc. FSE*, pp. 109–120. ACM, 2001.

[8] L. de Alfaro and T. Henzinger. Interface theories for component-based design. *Proc. EMSOFT*, LNCS 2211, pp. 148–165. Springer, 2001.

[9] L. de Alfaro, T. Henzinger, and M. Stoelinga. Timed interfaces. *Proc. EMSOFT*, LNCS 2491, pp. 108–122. Springer, 2002.

[10] D. Florescu, A. Grünhagen, and D. Kossmann. XL: an XML programming language for web service specification and composition. *Proc. WWW*, pp. 65–76. ACM, 2002.

[11] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. *Proc. ICWS*, pp. 738–741. IEEE, 2004.

[12] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. *Proc. CIAA*, LNCS 2759, pp. 188–200. Springer, 2003.

[13] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. *Proc. WWW*, pp. 621–630. ACM, 2004.

[14] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. *Proc. ICWS*, pp. 96–103. IEEE, 2004.

[15] X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web services. *Proc. CAV*, LNCS 3114, pp. 510–514. Springer, 2004.

[16] R. Hamadi and B. Benatallah. A Petri net-based model for web service composition. *Proc. ADC*, pp. 191–200. Australian Computer Society, 2003.

[17] J. E. Hanson, P. Nandi, and D. W. Levine. Conversation-enabled web services for agents and e-business. *Proc. IC*, pp. 791–796. CSREA, 2002.

[18] J. E. Johnson, D. E. Langworthy, L. Lamport, and F. H. Vogt. Formal specification of a web services protocol. *Proc. WS-FM*. Elsevier, 2004.

[19] S. Nakajima. Model-checking of safety and security aspects in web service flows. *Proc. ICWE*, LNCS 3140, pp. 488–501. Springer, 2004.

[20] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. *Proc. WWW*, pp. 77–88. ACM, 2002.

[21] M. Pistore, M. Roveri, and P. Busetta. Requirements-driven verification of web services. *Proc. WS-FM*. Elsevier, 2004.

[22] M. Solanki, A. Cau, and H. Zedan. Augmenting semantic web service descriptions with compositional specification. *Proc. WWW*, pp. 544–552. ACM, 2004.

[23] X. Yi and K. Kochut. A CP-nets-based design and verification framework for web services composition. *Proc. ICWS*, pp. 756–760. IEEE, 2004.

[24] J. Zhang, J.-Y. Chung, C. K. Chang, and S. Kim. WS-Net: A Petri-net based specification model for web services. *Proc. ICWS*, pp. 420–427. IEEE, 2004.